Last Modified: MJS May 9, 1983 1:38 PM
Filed on: {Indigo}<LOOPS>Loops-AAAI.bravo

Knowledge Programming in Loops: Report on an Experimental Course

by Mark Stefik, Daniel G. Bobrow,
Sanjay Mittal, and Lynn Conway

Knowledge Systems Area
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Abstract. Early this year fifty people took an experimental course at Xerox PARC on knowledge programming in Loops. During the course, they extended and debugged small knowledge systems in a simulated economics domain called *Truckin*. Everyone learned how to use the Loops environment, formulated the knowledge for their own program, and represented it in Loops. At the end of the course a *knowledge competition* was run so that the strategies used in the different systems could be compared. The punchline to this story is that almost everyone learned enough about Loops to complete their small knowledge system in only three days. The widespread surprise at this high level of productivity suggests that the power of integrating multiple programming paradigms has yet to be widely recognized.

1 Introduction

Knowledge programming is concerned with the techniques for representing knowledge in computer programs. It is important in many applications of AI, where the problems are messy. As in many situations in life, pat solutions and simple mathematical models just aren't good enough. Things break. Information is missing. Assumptions fail. Situations are complicated. To cope with messiness, AI researchers have found that large amounts of problem-specific knowledge are usually needed. This places a premium on the use of powerful techniques for representing and testing knowledge in computer programs.

Very few people have been trained to build knowledge systems. This is a critical bottleneck that limits the scope and impact of knowledge engineering. It limits the number of things that can be tried, the number of good ideas that are propagated, and the number of successful applications, that influence the way that other scientists and the general public perceive the field.

A few numbers may serve to put this in perspective. About one computer science researcher in ten does some work in AI, and perhaps a fifth of those work in knowledge engineering. In 1980, there were approximately 265 Ph.D. graduates in Computer Science, according to the "Snowbird Report" (Denning, et al 1981). Fewer than a half dozen doctoral theses appear each year on some aspect of building knowledge systems. An estimate in a brochure by Teknowledge, Inc., indicates that there are only about sixty people in the world with high level expertise in the design and development of knowledge systems. Although precise figures for these populations are difficult to obtain, all the evidence suggests that the community is tiny, indeed.

Our goal is to increase the impact and scale of knowledge engineering by simplifying the methods of knowledge programming and making them more widely accessible.

Training in knowledge engineering usually requires several years of study at one of a handful of universities. A group of us in the Knowledge Systems Area at Xerox PARC is trying to shorten this training time. Our goal is to increase the impact and scale of knowledge engineering by simplifying the methods of knowledge programming and making them more widely accessible. In doing this we have developed an experimental knowledge programming system called Loops -- (Bobrow & Stefik 1981), (Stefik, et al 1983a). Feedback about the adequacy of Loops is collected from beta-test sites which are using it to build knowledge systems. Feedback about the learnability of Loops is collected from the participants in an experimental course (see figure **xxx**) that we run periodically.

2 Integration and Paradigms

Knowledge programming is concerned with the techniques and paradigms for programming knowledge in computers. An important principle of knowledge programming is that different paradigms are appropriate for different purposes. This contrasts with the use of a single programming paradigm for everything, be it logic programming as in Prolog (Clocksin & Mellish 1981), lisp programming (Winston & Horn 1978), object-oriented programming as in Smalltalk (Goldberg & Robson 1983), or whatever.

An important principle of knowledge programming is that different paradigms are appropriate for different purposes.

There are various metrics of cost for applying a programming paradigm across a spectrum of applications. Examples of metrics are the cost of learning, the cost of modifying, the cost of debugging, and the cost of running. These costs vary across paradigms and applications because different programming paradigms provide different ways of organizing information in programs. For a given metric and application, some programming paradigms can be more cost-effective than others.

By allowing for choice and combination of paradigms, a knowledge programming system enables various costs to be lowered. For example, we attribute much of our success in creating a rapid introduction for building knowledge systems to the low costs for learning and applying Loops. For each of the things that the course participants needed to represent in their knowledge systems, there was *some* paradigm in Loops in which the expression of the knowledge was concise and the learning cost was low.

As indicated in the Loops logo in Figure 1, Loops currently integrates four programming paradigms:.

Procedure-oriented programming: In this paradigm, large procedures are built from small ones by the use of subroutines. Data and programs are kept separate. Most computer languages are like this. The procedure-oriented part of Loops is Interlisp-D (Teitelman 1978), (Xerox 1982). Interlisp-D is shown at the base of the Loops logo to suggest that it provides the solid foundation, on which the rest of Loops is built.

Object-oriented programming: In this paradigm, information is organized in terms of objects, which combine both instructions and data. Large objects are built up from smaller objects. Objects communicate with each other by sending messages. The conventions for communicating with an object with messages constitute a message protocol. Standardized protocols enable different classes of objects to respond to the same kinds of messages. Inheritance in a class lattice enables the specialization of objects.

Access-oriented programming: This paradigm is useful for programs that monitor other programs. Its basic mechanism is a structure called an *active value,* which has procedures that are invoked when variables are accessed. A useful way to think of active values is as *probes* that can be placed on the object variables of a Loops program. These probes can trigger additional computations when data are changed or read. For example, they can drive *gauges* that display the values of variables graphically.

Rule-oriented programming: This paradigm is specialized for representing the decision-making knowledge in a program. In Loops, rules are organized into *rulesets* which specify the rules, a control structure, and other descriptions of the rules. Two key features of the rule language are that it provides techniques for factoring control information from the rules, and also dependency-trail facilities, which provide a basis for "explanation" (see figure **xxx**) and belief revision.

Figure 1. **The Loops Logo.** This logo illustrates the different paradigms in Loops - procedure-oriented, object-oriented, data-oriented, and rule-oriented. Knowledge bases are used to provide facilities for long term storage and development of knowledge systems. The ring is intended to suggest that Loops *integrates* the paradigms. They are not just complementary. They are designed to be used together in building knowledge systems.

Each paradigm provides a vocabulary and a set of composition methods for organizing information in a program. These different organizational methods determine the way that information is factored and shared.

Procedure composition: The composition methods of Interlisp-D are forms of familiar control statements for iteration, recursion, and procedure call.

Object composition: This paradigm provides several composition methods as shown in figure **xxx**. The simplest approach is the specialization of methods and variables of a superclass. In addition, the inheritance *lattice* (in contrast with an inheritance *hierarachy*) enables inheritance to be factored. This allows the creation of special classes called *Mixins*, intended to impart a specific set of behaviors to subclasses. This terminology is borrowed from Flavors -- (Weinreb & Moon 1981). Another idea is *composite objects*. This idea extends the notion of objects to be recursive in structure so that multiple objects can be instantiated together. Finally, *perspectives* in Loops are groupings of objects into a higher level object, such that each component is a view (or perspective) of the whole. Perspectives provide for the forwarding of messages to the appropriate view.

Access composition: The main method of composition in this paradigm is the nesting of active values. Analogous to the use of multiple probes in measuring a circuit, this composition assumes that the "probes" are for independent instruments and do not interfere with each other.

Rule composition: The rule-oriented program provides for the sharing of rules among rulesets. It makes use of the other paradigms for organizing the interactions between the rules. Thus rules can call rulesets directly (using the procedural orientation), or invoke rulesets by sending messages (using the object orientation), or invoke rulesets by accessing data (using the data orientation).

Integration has two major themes in Loops: integration to allow the paradigms to be used together in building a knowledge system, integration of a programming environment for creating and debugging knowledge systems.

Some examples illustrate the integration of paradigms in Loops: the "workspace" of a ruleset is an object, rules are objects, and so are rulesets. Methods in classes can be either Lisp functions or rulesets. The procedures in active values can be Lisp functions or rulesets. This style is the meaning of the ring in the Loops logo, that Loops not only contains the different paradigms but integrates them. The point is that the paradigms are not only designed to complement each other, but also to be used in together in combination.

Figure 2. Combining paradigms: The perch approach.

Some examples illustrate non-integration of programming paradigms. For example, figure 2 shows the connection between Planner and Lisp. Planner was implemented in Lisp, but a program that used Planner would not call Lisp directly. Figure 3 shows the connection of List operations to Prolog. In effect, list operations were added late to Prolog after the initial design, but they have not been integrated in a coherent way with the database that underlies Prolog.

Figure 3. Combining paradigms: The patch approach.

Figure 4 illustrates another approach, illustrated perhaps by the Spice Machine. In this example Lisp and Pascal communicate, in a way, over a narrow bridge.

Figure 4. Combining paradigms: The bridge approach.

The second theme of integration is the integration of the programming environment. For example, Loops extends to other paradigms many of the facilities of Interlisp-D, such as the display-oriented break package, editors, and inspectors. In Loops this integration has led to the same synergy that is exploited in using multiple paradigms for application programs. For example, the notion of "breaking" on access to a function is extended to breaking on *access to a variable* by using active values to invoke the break package; the notion of tracing is extended to the notion of having gauges (see figure **xxx**) that can monitor the values of variables.

3 Getting Ready for the First Course

On January 6, we began to plan the first Loops course that would be offered on January 31 to our beta-test sites. We had a preliminary course outline, but we knew that we needed some way to draw the participants into programming in Loops. The idea of a video game was suggested, say rocket ships with Loops programs controlling the thrust and phasers. This idea was rejected as being both too frivolous, and computationally too expensive. Another suggestion was a game for placing tiles. We knew from (Malone 1980) that there were principles for making games motivating. Our course participants would be computing professionals drawn from research organizations and AI start up companies, who were interested in using Loops for building expert systems. We needed something that they would find useful and appealing.

As brainstorming continued, some pedagogical principles began to emerge. The game should draw on the real world knowledge of our students. Rocket ships and tiles were wrong, because people didn't have experience with such things from their everyday lives. A board game like Monopoly was considered, and then our first concept of *Truckin* emerged. It would be a board game with road stops. The players would drive trucks around buying and selling commodities. Their job would be to plan a route and make a profit. There would be various hazards along the way, places where goods and profits could be lost. Players would need to buy gas occasionally.

The best way to learn about knowledge programming in Loops is by extending a small knowledge system.

By mimicking real life, *Truckin* would provide the kinds of difficulties that knowledge engineers encounter in building expert systems. We could create a rich and animated simulation environment for the "independent truckers". The students would need to add knowledge to make their automated players more powerful. The simulation environment should draw on the student's real-world knowledge, and be rich enough to preclude a simple numerical model. Much of the appeal of this was that the "common experience" character of *Truckin* as a domain would enable us to side-step the usual knowledge acquisition bottleneck. The knowledge engineering experience would be accelerated by immediate feedback from the animated simulation. To simplify getting started, we would provide the students with a small expert system. We became convinced that the best way to learn about knowledge programming in Loops is by extending a small knowledge system.

The simulation environment should draw on the student's real-world knowledge, and be rich enough to preclude a simple numerical model.

At this point, we had less than a month to create the course materials, lectures, and *Truckin*. Sleep would become a rare and precious commodity. The *Truckin* data base began to take shape. The players would start at *Union Hall*, and would try to be parked at *Alice's Restaurant* at the end of the game. There would be various kinds of hazards of the road. The player with the most cash at Alice's would win.

Wonderfully, Loops was able to accommodate changes as our ideas evolved. Initially, we thought of the hazards as being road stops. This was probably a carry over of our childhood experiences with board games. Then we added the idea of "bandits" that could move around just like the independent truckers. Bandits were represented as an inheritance combination of players and consumers. We used active values on variables of the road stops to update the display for commodity prices and inventories. This meant that we did not need to find every place in the program where these things could potentially be changed, in order to update the display. The features of Loops worked for us, providing convenient techniques for factoring the program. We became experienced consumers of our own knowledge programming system as we raced to get ready for the course.

The simulation was designed to cause goal conflicts. A truck going quickly over a rough road would probably have its fragile merchandise damaged. A truck going quickly past a weigh station would probably get an extra fine, unless he was lucky or the weigh station was busy. On the other hand, a truck going slowly past a bandit would probably get intercepted. There would be perishable goods and fragile goods. We considered explosive goods and other such things, but removed them when they failed to add anything new to the game. Our pedagogical style was to leave some things out in order to keep it simple. A player could take only three kinds of actions: buy, sell, and move.

To facilitate the "suspension of disbelief" in watching the animated simulation, artistic attention had to be given to the appearances of things. Icons for the various commodities, hazards, and trucks were created. We experimented with different configurations of the gameboard, moving away from the outside edge configuration of most gameboards in order to pack enough road stops on the display screen. Highways were drawn next to the road stops, with a gray background and little dashed white lines in the center. People looked at intermediate versions of the gameboard and told us that the abrupt motion of the trucks was startling. We modified the code to simulate braking so that trucks would slow down as they arrived at their destinations. The visual appearance of *Truckin* became seductive. People were drawn into it.

The knowledge engineering experience is accelerated by immediate feedback from the animated simulation.

Prior to the this, we had used a simple gauge in our demo to illustrate the application of active values. It was a crude looking gauge and had little generality. We decided to extend the collection of gauges so that people could use them for debugging and for monitoring their independent truckers during the simulation. For ideas on style, we collected some professional catalogs of gauges, and sought advice from Bruce Roberts on the Steamer project. A family of gauges was designed (see figure **xxx**). Because of the extensive use of multiple inheritance and the interactions on the display between the parts of the hybrid gauges, a number of programming issues surfaced. The gauges went through several design reviews, to make the gauges simpler to use and modify. During these reviews, we created names for certain categories of design errors that we encountered. For example, a *grainsize* error is a situation where the structural parts of an object (usually methods) are factored too coarsely for the fine control needed by its specializations. A *replication error* is a situation where almost the same structure is repeated in parallel classes, instead of factoring it in a way that would allow it to be shared. Such experiences gave us a deeper understanding of the programming issues that people would encounter in using the different paradigms.

About two weeks before the course would begin, we sent out notices to our beta-test sites inviting people to sign up for the course. We expected about a half dozen people to sign up for the course. We advertised that our course would provide hands-on experience in extending a "mini-expert system". By word of mouth, the story spread. Over fifty people called us, requesting to get on the list. We split the list in half and scheduled the second course for the end of February. We didn’t send out any more advertisements. We had gone public and now we had to make it work.

Suddenly it was the weekend before the course. We made some guesses about the appropriate distribution of prices and penalties. We created our first automated player -- the *Traveler* -- which would just travel along the board between *Union Hall* and *Alice’s Restaurant*. As the *Traveler* cruised tirelessly around the game board, various bugs in the simulation surfaced. Meanwhile, we started work on a player to specialize in luxury goods called *HighRoller*. We didn’t have time to debug it very well before the course started. We reasoned that the bugs were acceptable, since they would provide things for the course participants to fix. We were right, but in hindsight, we had a lot of gall.

4 The Courses as Experiments

Since that weekend we have run two intensive knowledge programming courses, and also repeated the second course to a small group using videotape. By the time of publication of this article, the course will have been run for over 100 people. The courses are organized to alternate lectures and hands-on exercises. So far, everyone taking the course has learned enough about the Loops knowledge programming system to do some practice exercises (such as building a new kind of gauge) and extend a knowledge base for a *Truckin* player.

The most important aspect of the courses for our purposes is the opportunity that they provide us for refining both Loops and the course materials. For us, the courses are experiments, from which we are discovering how to make Loops and our teaching methods more effective. The basic structure of our experimental process is to run a course and to take some measurements. After the course, we change some parameters, run the course again and take the measurements again. Afterwards, we examine how the measurements differ and form hypotheses to guide the next iteration. The measurements are based on the performance of our students in terms of the problems that they complete and questions that they ask, and also on the results of questionaires that they return.

Between the first two courses we changed several parameters of the course, Loops, and *Truckin*:

- We substantially increased the emphasis on tools and techniques for debugging, providing heuristics for programming in Loops. Our concept of debugging is to provide tools for understanding the behavior of a system. The second course led students to use gauges for monitoring the values of variables, explanation facilities (figure **xxx**) for understanding which rule made a particular decision, and breaking and tracing facilities for discovering why some rules do *not* fire.

- In some cases, we introduced intermediate problems in the exercises. We hypothesized that some of the steps between exercises were too difficult to take all at once.

- We fashioned a new starting player for the second course, called the *Peddler*, which did a better job than *HighRoller* in factoring the concerns of an independent trucker. We hypothesized that the issues in restructuring *HighRoller* were too difficult for the three day course.

- We adjusted the prices and risks in the various commodities to provide a greater reward and selection pressure for more sophisticated and knowledgeable truckers.

- We improved the browsers, that is, our interactive graphics for "browsing" information in a knowledge base (see figure **xxx**). We believed that we could reduce much of the cognitive load for restructuring objects and accessing information if we provided more effective ways of making the right information visible.

- We fixed troublesome bugs in the rule compiler. During the first course, participants had to struggle with a compiler that did not reliably keep the generated Lisp code in correspondence with the rules.

The weakest player from the second course could easily dominate the best player from the first course.

As a result of these changes, the participants in the second course were dramatically more successful than those in the first. In the first course, we had to slip the schedule for the knowledge competition by 90 minutes, in order to let people finish preparing their players. In the second course, people had players ready in about half of the allocated time, and spent the remaining time exploring other aspects of the system and tuning their strategies. Furthermore, the weakest player of the second course could easily dominate the best player from the first course.

People asked far fewer questions in the second course, and were able to complete many more of the exercises. In addition, the questionaires from the second course came back with radically different advice from those from the first course. The general response from the first course was "give us less on rules" and many people indicated substantial concern with many of the fundamental aspects of that paradigm. In the second course, the responses turned completely around. They said "give us more on rules and debugging".

We hypothesize that in the first course the combination of a faulty rule compiler and lack of information on how to debug programs in this paradigm undermined confidence. During the second course, two members of a team were observed staring at a display. One of them said, "why is it buying tomatoes?", and the other one elbowed him saying "Ask why! Ask why!" -- goading him into action at the Loops keyboard. They had learned their lessons well.

This process of tuning the course and Loops in response to feedback reflects our interest in methods for the *engineering of knowledge* (Conway 1981), (Stefik & Conway 1982). In this case we are engineering languages and techniques for knowledge programming. The courses provide a source of feedback on the effects of changes that we make to the course materials, paradigms and programming environments. We expect this evolutionary process to yield further optimizations in Loops --- improving further the ease of learnability and use. In time, we would like to extend our work to provide a framework that would simplify the process of creating higher level organizations in expert systems (Stefik *et al* 1982).

5 The Knowledge Competition

One of the most rewarding things about the Loops course is the kind of electric excitement that erupts during a knowledge competition. People seem to project themselves into the players that they have created. They have put their player through many simulations and many playing conditions. In a sense, they have taught it everything. But during the competition there is a moment of truth. The rules cannot be changed. Success in the short run is affected by chance, but on average, the most knowledgeable players will win.

The randomly generated game board comes up. As the simulation begins, there is a great deal of commentary and jibes as people compare their players. Who's ahead? Who just got robbed? A wonderful thing about *Truckin* is that the silliness of the ill-fated move is something that all the observers appreciate almost immediately. For example,

o A player may be racing to *Alice's Restaurant. O*ne move before the game ends, and it is unable to resist a business "opportunity" and doesn't make it to *Alices*.

o A player may go to the closest place to sell some goods, even if it happens to be the *City Dump*, which unfortunately pays a "negative price".

o A player may get focused on a tight producer/consumer loop, making money faster than any other player on the board. If it is programmed to buy fuel from stations that it encounters along its route, but there is no gas station in the tight loop, the team watching the competition will watch dismally as the fuel gauge drops lower and lower. Eventually the truck runs out of fuel and gets towed back to *Union Hall* where it must start afresh.

o A player may try to park next to *Alice's Restaurant* near the end of the game, even if that happens to be the *Union Hall*, which confiscates all goods and cash.

In our experience so far, these oversights happen in the best of players. They provide a source of merriment during the competition, and an illustration of just how much knowledge is really needed to be powerful, even in an artificial environment.

Success in the short run is affected by chance, but on average, the most knowledgeable players will win.

The knowledge competition also serves as a source of examples and metaphors about the nature of knowledge. One example drawn from the first Loops course illustrates the interplay between knowledge and environment. For the first knowledge competition, two teams prepared players by simply fixing some of the bugs in the *HighRoller*. They had a private playoff just before the competition, and discovered that when both players were in the same game, the inventory of luxury goods on the game board became exhausted before the end of play. Neither player was able to cope with this situation. One of the heuristics that we now offer to teams preparing for a knowledge competition is to test rules with many copies of the same player competing at once.

This interplay between knowledge and environment brings to mind the example of the ant on the beach (Simon 1981), in which the apparently complex movement of the ant on the beach is attributed to the complexity of the beach environment rather than the complexity of the ant. In *Truckin*, the "ants" are mechanical and programmable. We have observed that even the complexity of the *Truckin* environment creates a substantial selection pressure for resourceful and knowledgeable players. To win, the designers of the players must pit knowledge against complexity. Knowledge provides the adaptability needed for mastering the situations in the game.

The primary example of metaphors from the knowledge competition inspired the name of the event. This is the observation that it is truly the knowledge of players that is competing, and the most adaptable player wins. Recently in connection with the interest in fifth generation computers, Feigenbaum and McCorduck (1983) have characterized knowledge as the new "wealth of nations". In the knowledge competition and *Truckin*, the competitive advantages of knowledge in a player is concrete and observable in short experiments.

The success of the knowledge competition in motivating participation has led us to speculate on ways of alleviating the knowledge acquisition bottleneck by triggering participation in a community of experts. One idea is to create *knowledge servers*, which accept knowledge over a computer network and make themselves available for solving problems. Here again there would be a "competition" between different bodies of knowledge, competing to solve the problems that are posed.

6 Implications

Sometimes the effects of a technological change can be surprising and widespread. We sense that a technological change is emerging from research on knowledge programming, a change in the infrastructure for building knowledge systems. This is not to claim that Loops has reached its final form, or even that Loops will necessarily be the system in which the effects are first widely spread.

The general technological shift is the simplification of techniques for knowledge programming. The shift will have leveraging power in two ways: (1) the freeing of existing knowledge engineers from spending a year or two building the bottom of their knowledge representation systems and (2) a measurable acceleration in the progress of the field if the simplified methods trigger an increase in the number or practitioners from 100 to 1000 or more. Knowledge engineering would then begin to have a noticeable effect in many areas of our lives.

Acknowledgments

We extend thanks to x, y, and z for advice and comments on an earlier drafts of this paper. Thanks especially to the participants from Applied Expert Systems, Daisy Systems, ESL, Fairchild AI Lab, Lawrence-Livermore Laboratories, Schlumberger-Doll Research Laboratory, SRI International, Stanford University, TeKnowledge, and Xerox Corporation for the many helpful suggestions during the first two courses, and for the pleasure we had working together and observing the knowledge competitions.

Note: The Loops course is offered periodically by the Knowledge Systems Area at Xerox PARC. Loops is being made available to selected Xerox customers that have been designated as *beta-test* sites. Planning is currently underway to consider the possibility of developing and supporting Loops as a product, and licensing it to other computer vendors.

Need KA Bottleneck reference.

Steamer Reference

Bibliography and Further Reading

Bobrow, D. G. & Stefik, M. (1981) The Loops Manual. Tech. Rep. *KB-VLSI-81-13*, Knowledge Systems Area, Xerox PARC.

Clocksin, W. F. & Mellish, C. S. (1981) *Programming in Prolog*. Berlin: Springer-Verlag.

Conway, L. (1981) The MPC adventures: Experiences with the generation of VLSI design and implementation methodologies. *Proceedings of the Second Caltech Conference on Very Large Scale Integration*, 5-28.

Denning, P. J., Feigenbaum, E., Gilmore, P., Hearn, A., Ritchie, R.W., & Traub, J. (1981) The Snowbird Report: A discipline in crisis. *Communications of the ACM*, 24:370-374.

Feigenbaum, E., & McCorduck, P. (1983) *The fifth generation*. Reading, Mass.: Addison-Wesley.

Goldberg, A., & Robson, D. (1983) *Smalltalk-80: The language and its implementation.* Reading, Mass.: Addison-Wesley.

Malone, T. W. (1980) What makes things fun to learn? A study of intrinsically motivating computer games. Technical Report CIS-7 (SSL-80-11), Xerox Palo Alto Research Center.

Simon, H. A. (1981) *The sciences of the artificial*, Cambridge, Mass.: The MIT Press.

Stefik, M., Bobrow, D., & Mittal, S. (1983) Knowledge programming in Loops: Highlights from an experimental course. Videotaped Report KSA-83-1, Xerox Palo Alto Research Center.

Stefik, M., Bell, A. G., & Bobrow, D. G. (1983) Rule-oriented programming in Loops. Tech. Rep. *KB-VLSI-82-22*, Knowledge Systems Area, Xerox PARC.

Stefik, M., & Conway, L. (1982) The principled engineering of knowledge. *AI Magazine* 3(3):4-16.

Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., & Sacerdoti, E. (1982) The organization of expert systems: A tutorial. *Artificial Intelligence* 18:135-173.

Teitelman, W. (1978) Interlisp reference manual. Tech. Report, Xerox Palo Alto Research Center.

Weinreb, D., Moon, D. (1981) *Lisp Machine Manual*. Cambridge, Mass: MIT.

Winston, P.H. & Horn, B. K. P. (1981) *Lisp*. Reading: Addison-Wesley.

Xerox Corporation, (1982) *Interlisp-D users guide*. Pasadena, CA: Xerox Electro-Optical Systems.

More Figures to be placed and numbered.

Figure x. **Loops Course Outline**. This three-day course is offered periodically by the Knowledge Systems Area at Xerox PARC.

Figure x. **Object Composition in Loops**. The inheritance lattice enables many forms of structural sharing in Loops. The simplest form of specialization (shown at the left), is to create a subclass the overrides and augments variables and methods inherited from the superclass. *Mixins* is a term for a special kind of class intended to be used as a generic superclass in the system, conferring a set of capabilities on subclasses. When multiple superclasses are used, the resulting subclass is essentially a *hybrid*. Instances of the subclass mix together the attributes from the superclasses. *Perspectives* provide a way of grouping objects to act as views of a higher level object. It provides for the automatic forwarding of messages to the appropriate perspective.

Figure x. **The Loops Gameboard -- played by small knowledge systems called *independent truckers***. The little squares on the board are road stops, connected by the highway drawn above. They are producers, consumers, rough roads, weigh stations. Roadstops with icons are producers, where players can buy. Those with words (e.g., *Clothing*) are consumers, where players can sell. The trucks for the players are shown parked or moving along the highway (e.g., Sanjay). To the right, a panoply of gauges is being used to monitor the status of various players. In the upper left corner, a rule for one of the players is being traced.

Figure x. **Seeing the Knowledge behind a decision**. In this figure the game is interrupted, causing the Rule Exec window to pop up. The user has asked the system why his truck picked a particular stopping place, and Loops has displayed the rule that made the decision.

Figure x. **Class Browser on Commodities**. Browsers are interactive programs used to *browse* through a knowledge base. The lines in a class browser indicate superclass relationships. For example, in this figure, a *StereoSystem* is a *LuxuryGood*, an *Appliance*, and a *FragileCommodity*. Browsers can be created to show other relationships too. By selecting nodes in a browser with a mouse, a user can access further information.

Figure x. **Loops Gauges**. Gauges are debugging tools used to monitor the values of variables. They can be thought of as probes, that can be inserted onto the variables of an arbitrary Loops program. Gauges are defined in Loops as classes, and driven by active values -- the mechanism behind data-oriented programming in Loops. A browser at the bottom of the figure illustrates the relationships between the classes of gauges. From this figure, we can see that the *DigiMeter* is a combination of a *Meter* and an *LCD*.